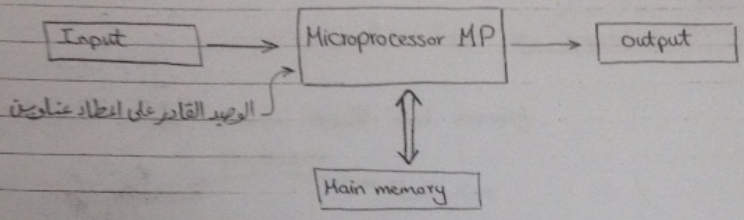
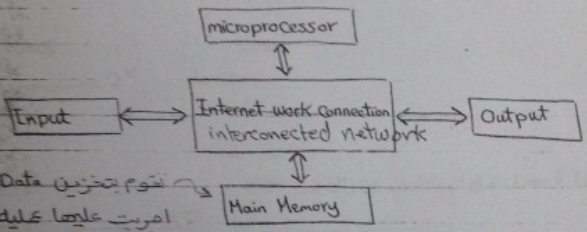
Ch 1 Microprocessor (Introduction)



* ——— *



Control processing Unit (cpu) ≡ Machine

→ cpu components

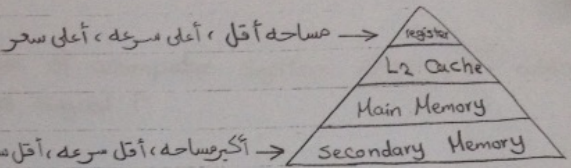
1. ALU
2. CU (Control Unit) تتحكم في وظائف الجهاز
3. Registers level 1 cache

→ cpu functions:

1. Execode arithmatic & logic instruction
   (Fetch - decode - execode)

→ Manage the program flow (make decisions)

* Zero * negative * overflow ...



→ Transfer Data between itself and memory

processor

الشغال → Multi-processor ⟷ Multi-Core

Buses: group of Common wires connect between parts of device.

1] Address Bus : →

K-bit ⟶ 2^K devices.

2] Data Bus : ⟷

كلما زاد كلما زاد معدل نقل البيانات من والى الـ MP ←

devices ⟷ MP

3] Control Bus : ⟷

ينقل Control Signal ←

devices ⟷ MP كلما زاد كلما كان أفضل

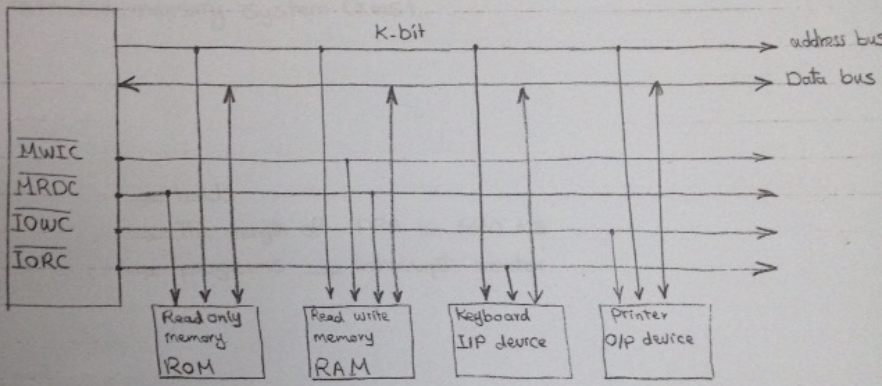
←

1-

2-

3-

Draw the block diagram of a computer system showing the address, data and control bus signal ?



* Simple Control Signal

MWIc	memory write control signal
MRDC	memory read control signal
IOWC	I/O write control signal
IORC	I/O read control signal.

→ MP → device التعامل بين

1- MP ⇝ send address

2- Control Signal MRDC

3- Data transfer.

The Memory System is divided into "3 main points"

1- Transient program area. (TPH)
2- System Area
3- extended memory System (xms)

[1] → holds
→ The length of TPA is 640 KB
→ programs → Interrupt vector

Ch2 : The Microprocessor & Its Architecture

8 bit	16 bit	8 bit	
AH	AX	AL	Accumulator
BH	BX	BL	Base Index
CH	CX	CL	Count
DH	DX	DL	Data
	SP		Stack Pointer
	BP		Base Pointer
	DI		Destination Index
	SI		Source Index

IP	
FLAG	

CS	Code Segement
DS	Data Segement
ES	Extra Segement
SS	Stack Segement
FS	
GS	

special purpose.

Programming Model Of The Intel 8086 Pentium 4

32 bit		8bit	16bit	8bit	
EAX		AH	AX	AL	Accumulator
EBX		BH	BX	BL	Base Index
ECX		CH	CX	CL	Count
EDX		DH	DX	DL	Data
ESP			SP		Stack Pointer
EBP			BP		Base Pointer
EDI			DI		Destination Index
ESI			SI		Source Index

	IP	Instruction Pointer
	FLAGS	Flags

CS	Code
DS	Data
ES	Extra
SS	Stack
FS	
GS	

The Programming Model Of The Intel 80386 Pentium 4

كل 4 bit في الشاشة يعطي 1 bit في السادس عشر ←

20 bit → (XXXXX) Hexa.

Real Mode

Segement	Off Set	Special Purpose
CS	IP - EIP	Instruction address
SS	① SP or ② BP , ESP / EBP	Stack address
DS	BX . DI . SI 8 bit 16 bit } numbers	Data address
ES	DI , EDI For string Instructions	String destination address
FS _ GS	No default	General address.

Programming Model

program Visible
- They are used during app-prog
- Ex: EAx . Ax . AI . EBX ...
  → Mov Ax , Bx

program invisible
- not addressed directly during app-prog
- only 80286 - P4 Contain this type
- ex: FIAG . EFIAG

SUB Ax . BX بعد تنفيذ أمر

Purpose Of Use

General purpose
EAX , EBX
Hold Data address or off set +
البعد عن البدايه

Special purpose.
CS, DS, ES, SS

int X = 3
address ↑ data ↑

Addressing modes (How the memory is accessed)
Addressing mode ← كثيره لكنها طرق تنظيم البيانات داخل الذاكره
وظيفتها الوصول إلى الذاكره

1- Real mode 1 MB only
2- Protected mode upto 4GB

« Real mode »
→ Accumulator EAX . AX . AL . AH
Can be used as 32bit → EAX , 16 → Ax , 8bit → AH, AL

1 Special purpose :
← طرف أساسى فى عمليات الضرب والقسمة
→ Div Ax , Bx : $\frac{Ax}{Bx}$

Ax : يوضع به الجزء الصحيح
DX : (Data register) يوضع به باقى القسمه "الكسر"
→ Mux Ax(16) , Bx(16)
16 * 16 ⟶ 32 bit

→ Mux EAX , EBX
32 , 32

DX high bytes | AX low bytes
EDX high bytes | EAX low bytes

Base Index

Can be used as 32 bit → EBX , 16 bit → BX

8 bit → BH , BL

General purpose → hold off set For Data segement

Note : In Real Mode

We have 64 KB processor , 5 bit memory

→ To access the memory للوصول للذاكره

← processor من الـ

$64K = 2^6 \cdot 2^{10} = 12^{16}$

$(16\ bit)_B \rightarrow (4\ bit)_{Hexa.}$

~> From 4 bit processor how to reach 5 bit memory

we add zero From the left side to Base

→ to reach to any address in the memory we need Base and off set

البدايه Base is in Code Segement

$$\text{Current address}_{(real)} = (CS)_0 + \text{off set}$$

Ex: CS = 4000 Hex , IP = 2342

Solution

Current address = $(cs)_0 + IP = 40000 + 2342 = 42342$ Hex

Count

To Count in any process with loop we need big register

loop Cx , ECX

Repetition Cx

Shift CL we need small number of loops

Data
Can be used as 32 bit → EDX 16 bit → DX
8 bit → DH, DL
General purpose → Div, Mux الضرب والقسمه

Stack pointer
Can be used as 16 bit → SP
hold the off set of stack segement (SS) ← الأولويه له

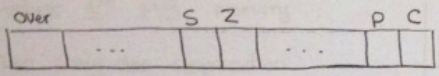
Base pointer
Can be used as 16 bit → BP
used as off set for stack segement (SS)

Destination Index
Can be used as 16 bit → DI له الأولويه بعد BX
hold the off set of Data Segement

Source Index
Can be used as 16 bit → SI له الأولويه بعد DI
hold the off set of Data segement

Instruction Pointer
Can be used as 16 bit → IP
hold the off set of current instruct to be excuted (in Code segement)

Flag . Invisible



Can be used as 16 bit - 32 bit

→ Help microprocessor to take the correct way

1 Over Flow happened when → +ve + +ve = -ve
→ -ve + -ve = +ve

2 Carry (C)

```
   11111010
Carry  11011110+
(1) ← 11101000
```

3 Parity (P)

1 → number of one's is even

0 → number of one's is odd

تستفيد من هذا عندما يتم نقل Data من مكان لآخر نتأكد من أن Data تم نقلها بدون فقد حيث اذا كان عدد 1's زوجي بعد النقل وكذلك P=1 فان النقل تم بالصوره المطلوبه والعكس يكون فيه فقد في Data

Ex: SS = 3245 H

BP = 2342

Sol

Base address = 33450 H

Current address = 33450 + 2342 = 35792

Ending address = Starting + FFFF

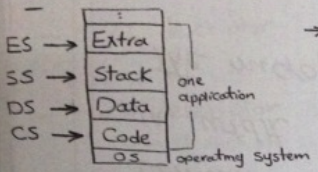
= 33450 + FFFF

Protected Mode

Processor $\xrightarrow[2\ modes]{access}$ Memory

[1] Real Mode

- access 1st MB memory
- run Only one application
- DOS



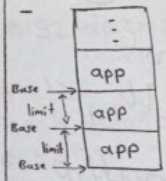
→ How to access memory?

- EA = (Base)$_0$ + offset

Base (starting)	&	offset
DS	→	Bx, DI, SI
SS	→	SP, BP
ES	→	DI
CS	→	IP

[2] Protected Mode. (80286 and above)

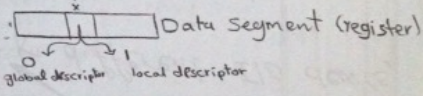
- access up to 4GB
- run more one application.
- GUT windows, linux



→ every application has different size from the other

→ How to access memory?

1- Selector : Choose one of descriptors



2- Descriptor

- each descriptor 8 byte.
- There are 8192 descriptor for every application
- → local Descriptors : describe local applications
  → Global Descriptors : describe global applications.
- Knows us ① Base : application start in memory
  ② limit : the depth of the end from base
  → end = Base + limit
  ③ access right → a- Code
  b- data → Read only / Read / write
  c- perivity

Application

- **Global (System) Application**
  - → The data of this type is public.
- **Local (User) Application.**
  - → The data of this type is private.

Descriptors

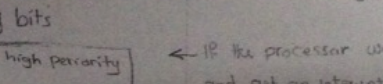
Global Descriptor	Local Descriptor.
— describe Global applications	— describe local applications.
— 8192 descriptor (number)	— 8192 descriptor (number)
— each descriptor 8 byte (size of each one in memory)	— each descriptor 8 byte (size of each one on memory)
— Total size $= \frac{8 \text{ byte} * 8192}{1024} = 64 \text{ KB}$	— Total size $= \frac{8 \text{ byte} * 8192}{1024} = 64 \text{ KB}$
— each descriptor describe • Base • limit • access right	— each descriptor describe • Base • limit • access right

Processor ← هناك سوى registers في بصيل اي الذاكره

وبالتالي يكون الاختلاف بين real mode و protected mode

في طريقة تفسير مايدا خل registers حيث في real mode

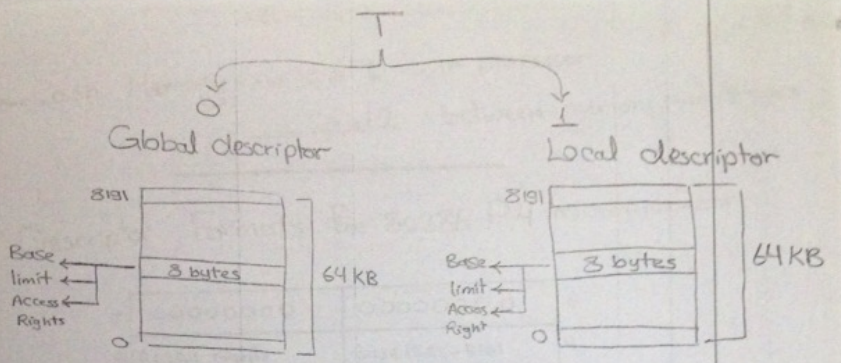
و في protected mode كالتالي $EA = (Base)_0 + offset$



① Periority bits

0 0	← high perority
0 1	
1 0	
1 1	← laws periority

← If the processor work with any program with 00 periority and get an interupt from another program with 01 periority the processor will still work with the first program but if the 1st program is with 11 periority and get an interupt the



We Know ① the type of descriptor (global - local)
② the number of descriptor (one of 0 to 8191)
③ Periority bits

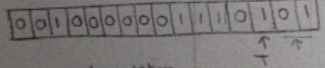
From Register

We Can Know ① Base ② limit ③ Access right
From Descriptor ← which we get from register.

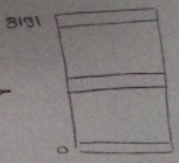
السؤال: أين تقع جداول descriptor؟

Example:

CS = 2075 H



T=1 local descriptor



① لوقت processor:

② لوقت Memory: كيف وأنا أستخدم الوصول الى الذاكرة يجب ان هنا access الذاكرة مرتين

③ Cash Memory: ذاكرة بين main memory و processor لتسريع عمل processor ومساعدتها صغيرة توضع بها البيانات المذكورة ولكنها هنا هي الحل الأمثل حيث يوضع بها الاجزاء المذكورة

→ Cash Memory → level 1 : in processor

→ level 2 : between memory and processor

The Descriptor Formats For 80286 P4 microprocessor

7	00000000	00000000	6
5	Access rights	Base (B23 – B16)	4
3	Base (B15 – B0)		2
1	limit (L15 – L0)		0

The Descriptor Formats For 80386/80486/P/PPro/PII microp-

7	Base (B31 – B24)	G	D	0	A V	limit (L19 – L16)	6
5	Access rights	Base (B23 – B16)					4
3	Base (B15 – B0)						2
1	limit (L15 – L0)						0

Ch3 : Addressing modes (Real Mode)

[1] Immediate Addressing Mode

I → R (I: information, R: register)

Examples:
- Mov BL, 44 , Mov BH, 44H
- Mov CL, 10011110 B
- Mov ECx, FC23F34EH

[2] Register Addressing Mode.

R ↔ R (register ↔ register)

- Mov CL, BL (✓)
- Mov Cx, BX (✓) | Mov CS, BX (x)
- Mov EAX, EBX (✓)
- Mov Ds, Cx (✓) | Mov BX, CS (✓)
- Mov Ds, CS (x)

[3] Direct Addressing Mode

R ↔ M (memory)

- Direct addressing
  - R: EAX, AX, AH, AL
- Displacement (off set) addressing
  - Any Register.

Direct	Displacement
Mov AL, Num	Mov CL, Num
Mov Ax, Num	Mov Cx, Num
Mov Num, AX	Mov EBX, Num
Mov EAX, Num	Mov Dx, [1023]

[4] Register Indirect Addressing Mode[]

(BP , Bx , SI , DI)

R ↔ [I]

- Mov Ax, [Bx]
- Mov [BP], DL
- Mov EDX, [DI]
- Mov [BX], [SP] (X)

[5] Bass_plus_Index Addressable Mode

EBX → Bx, EBP → BP

ESI → SI, EDI → DI

use with one dimention array

R ↔ M

- Mov Dx, [Bx+DI]
- Mov WORD / DWORD PTR [BX+DI], DX

[6] Register_Relative Addressable_Mode

BP, Bx, SI, DI / EBP, EBX, ESI, EDI

off set

مكان الذاكرة

R ↔ M

- Mov Ax, [BX+1000]
- Mov Ax, 1000[DI]
- Mov WORD / DWORD PTR [Bx+1000], AX

[7] Base_relative plus Index Addressable Mode

Used in two dimention Array

R ↔ M

- Mov WORD PTR 1000[BX+DI],
- Mov DH, [BX+DI+20H]

Effective Address (EA)

No EA

Notes

( )10 عشرى

( )B Binary ثنائى

( )H السادس عشر ، ' ' Characters

R_{source} (R_S)

الذى نحصل منه على البيانات Register

R ↔ R has the same size 8↔8 16↔16

← لا يجوز بأى حال من الأحوال تغير Code segment (CS) الذى يحتوى على عنوان الأوامر داخل الذاكرة وكذلك لا يجوز النقل منه segement الى آخر

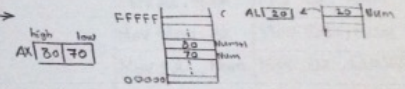
E.A = memory = Num

⇒ EA هو المكان الذى نضع به أو نأخذ منه البيانات ولا يتأثر بنوع Instruction أو معنى أمر لا يتأثر بهذه

1→ Memory ⇝ Byte addressable

Byte by Byte متوقعه

2→



little endian assignement

$EA = (seg)_{+0} + offset$

① $EA = (DS)_0 + BX$

② $EA = (SS)_0 + BP$ ③ $EA = (DS)_0 + DI$

عند نقل بيانات من Register الى Memory لابد من تحديد المساحه المطلوب تخزين البيانات بها حيث أن Memory byte addressable

Mov [BX], AL ← يتم تحديد مساحه التخزين

يتم التعديل الى ←

Mov Byte PTR [BX], AL

Mov WORD PTR [BX], AL

Mov DWORD PTR [BX], AL

$EA = (DS)_0 + BX + DI$

Starting address

$EA = (DS)_0 + BX + 1000$

يظل effective address ثابت

$EA = (DS)_0 + \begin{matrix} BX \\ DI \\ 1000 \end{matrix}$

عند نقل بيانات من Register الى Memory لا تحدد المساحه للتخزين ولا توجد مشاكل

Ch 3 : Addressing modes (Real Mode)

		Examples
[1] Immediate Addressing Mode	I → R (I: information, R: register)	Mov BL, 44 , Mov BH, 44H Mov CL, 1001110 B Mov ECX, FC23F34EH
[2] Register Addressing Mode.	R ↔ R (register ↔ register)	Mov CL, BL (✓) Mov CX, BX (✓) Mov EAX, EBX (✓) Mov DS, CX (✓) Mov DS, CS (X) Mov CS, BX (X) Mov BX, CS (✓)
[3] Direct Addressing Mode Direct addressing ↓ R: EAX, AX, AH, AL Displacement (offset) addressing → Any Register.	R ↔ M (memory)	Direct: Mov AL, Num / Mov AX, Num / Mov Num, AX / Mov EAX, Num Displacement: Mov CL, Num / Mov CX, Num / Mov EBX, Num / Mov DX, [1023]

[4] Register Indirect Addressing Mode [ ]
↳ (BP , BX , SI , DI)
	R ↔ [M]	Mov AX, [BX]
		Mov [BP], DL
		Mov EDX, [DI]
		Mov [BX], [SP] (X)

[5] Bass_Plus_Index Addressable Mode EBX → BX, EBP → BP, ESI → SI, EDI → DI use with one dimention array	R ↔ M	Mov DX, [BX+DI] Mov WORD / DWORD PTR [BX+DI], DX
[6] Register_Relative Addressable_Mode. BP, BX, SI, DI / EBP, EBX, ESI, EDI offset مكان الذاكرة	R ↔ M	Mov AX, [BX+1000] Mov AX, 1000 [DI] Mov WORD / DWORD PTR [BX+1000], AX
[7] Base_relative plus Index Addressable Mode Used in two dimention Array	R ↔ M	Mov WORD PTR 1000 [BX+DI], A Mov DH, [BX+DI+20H]

Effective Address (EA)

No EA

Notes

( )$_{10}$ عشرى

( )$_{B}$ Binary ثنائى

( )$_{H}$ السادس عشر , ___ , Characters

R_{source} (R_s)

الـ Register الذى نحصل منه على البيانات

$R \leftrightarrow R$ has the same size (8 ↔ 8, 16 ↔ 16)

← لا يجوز بأى حال من الأحوال تغير Code segment (CS) الذى يحتوى على عنوان الـ IP وامر داخل الذاكره
ولذلك لا يجوز النقل منه الى أى segement آخر

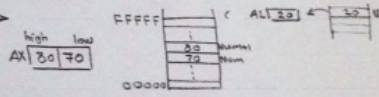
E.A = memory = Num

⇒ EA هو المكان الذى نضع به أو نأخذ منه البيانات ولا يتأثر بنوع Instruction أو بمعنى أصح لا يتأثر بما قبله

1→ Memory ⇝ Byte addressable

Byte by Byte مترقمه

2→



little endian assignement

$EA = (seg)_{+0} + OFFset$

① $EA = (DS)_0 + BX$

② $EA = (SS)_0 + BP$ ③ $EA = (DS)_0 + DI$

$EA = (DS)_0 + BX + DI$

Starting address

$EA = (DS)_0 + BX + 1000$

$EA = (DS)_0 + \begin{matrix} BX \\ DI \\ 1000 \end{matrix}$

← عند نقل بيانات من Register الى Memory لابد من تحديد المساحه المطلوب تخزين البيانات بها حيث أن Memory byte addressable

Mov [BX], AL ← لم يتم تحديد مساحه التخزين
لذا يتم التعديل الى ←

Mov Byte PTR [BX], AL

Mov WORD PTR [BX], AL

Mov DWORD PTR [BX], AL

ويظل effective address ثابت

← عند نقل بيانات من Memory الى Register لا تحدد المساحه للتخزين ولا توجد مشاكل

Ch 4 : Data Movement Instructions

1- Assembler & Deassembler

2- Stack (push, pop)

3- Instructions

Assembler & Deassembler

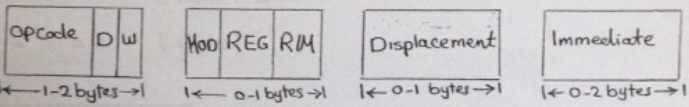
Hinit : Assembler : transform the assembly code to machine code

Deassembler : transform the machine code to assembly code

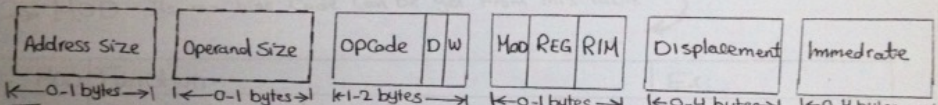
→ machine code (0's, 1's)

↝ Modes of Instruction

[1] 16 bit Instruction mode (8086)



[2] 32-bit Instruction mode (80386 above)



To transfere from assembler to deassembler follow this steps

[1]

Operand Size	Address Size	mode
66 H X	67 H X	- 16-bit instruction mode with its default (8 - 16 bit) registers w=0 w=1
✓	X	- 16 bit instruction but works with (8 - 32 bit) registers. w=0 w=1
✓	✓	- 32 bit instruction works with its default (8 - 32 bit) register w=0 w=1
X	✓	- 32 bit instruction but works with (8 - 16 bit registers. w=0 w=1

[2] | Opcode | D | W |

→ Opcode: has machine code of the instruction (Mov, sto, swap)
we get this code from the table
like Mov ⟶ 100010

→ D ⇝ D=1 (when we transfer from Memory to register)
⇝ D=0 (when we transfer from register to Memory)

→ W ⇝ W=1 (word / Dword)
⇝ W=0 (Byte)

[3] | MOD | REG | R/M |

→ MOD : Its machine code can be get from this table ⤵

MOD	Function		Ex:
00	No Displacement	R/M is memory	Mov AL, [DI]
01	8-bit sign-extended displacement		Mov AL, [DI+2]
10	16-bit displacement		Mov AL, [DI+1000h]
11	R/M is a register		

→ REG : Its machine code can be get from this table

Code	W=0 (Byte)	W=1 (Word)	W=1 (Dword)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Code	Seg
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS

→ R/M ⇝ to get the code of R/M → at first we want to know if R/M is Memory or Register that depend on MOD

IF MOD = 11 ∴ R/M is registers
we can get its code from register table.

code	W=0 (Byte)	W=1 (word)	W=1 (Dword)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

IF MOD = (00, 01, 10) ∴ R/M is Memory
we can get its code
from this table

R/M code	Addressing mode	Function
000	DS:[BX+SI]	DS:[EAX]
001	DS:[BX+DI]	DS:[ECX]
010	SS:[BP+SI]	DS:[EDX]
011	SS:[BP+DI]	DS:[EBX]
100	DS:[SI]	Uses scaled-index byte
101	DS:[DI]	SS:[EBP]*
110	SS:[BP]* special addressing mode	DS:[ESI]
111	DS:[BP]	DS:[EDI]
	16-bit R/M memory addressing mode	32-bit addressing mode selected by R/M

④ Displacement

① 1 byte □□ ⇝ Put as it

ex: 1 → 01 , 21 → 21

② 2 byte □□ □□ ⇝ Swap high, low

ex: 1234 → 3412

ex: 123 → 2301

Stack
(pop & push)

→ The PUSH and POP instructions are important instructions that store and retrieve data from the LIFO (last In First Out) stack memory.

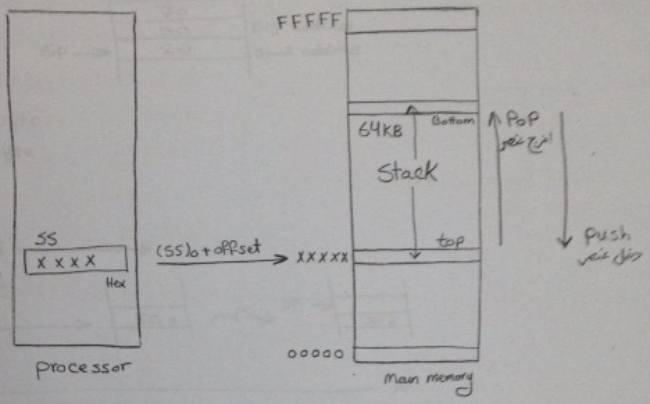
Stack (LIFO)

↓

two operations

Pop — يخرج عنه ويرد pointer

Push — Pointer يحرك ويدخل عنصر

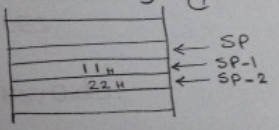


عندما Stack بالعناصر ووضع عنصر آخر يحدث Cycle nature اي يتم دفع ما بالعنوان الاول ووضع العنصر الجديد في مكانه

→ Push AX (✓)

AX: 11H | 22H (high byte | low byte)

Stack في AX وضع محتوى



→ Push 5262H (✓)

→ Push [BX] (X) ← هنا لم يحدد المساحة التي يخزن بها البيانات في memory

EA = (DS)o + BX

The Correct

Push Byte PTR [BX] (✓)

Push (word - Dword) PTR [BX] (✓)

→ Push CS (✓) → هنا لم يغير محتوى CS

* Hinit

22H Can be Put in memory as

1 Byte: 22H

2 word: 00 / 22

3 Dword: 00 / 00 / 22

is performed on bytes, words or doublewords,

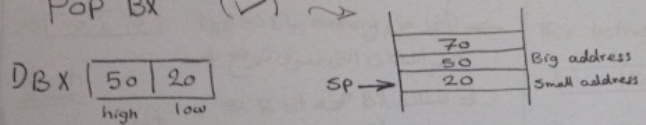
to - [illegible]

Pop ← stack of data

Pop 2345 H (X)

Pop CS (X) → we mustn't change CS

Pop BX (✓) →



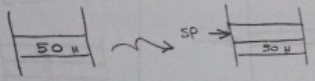
high address → high byte

low address → low byte

② Then SP = SP + 2

Pop Byte PTR [BX]

EA = (DS)₀ + BX →



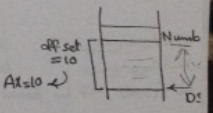
Instructions

- LEA
  - LDS, LSS, LFS, LGS, LES
  - * Off set
- LODS, STOS, MOVS, INS, OUTS.
  - * IN, Out
  - * X ChG
  - * BSwap
  - * CMOS

→ [1] LEA (load effective address)

ex: LEA AX, Numb ≡ Mov Ax, offset Numb

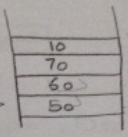
هذا الأمر معناه وضع بعد المكان Numb عن البداية في AX



ex: LEA BX, [DI] → EA = (DS)₀ + DI ≠ Mov BX, offset [DI] (X)

→ LDS

ex: LDS BX, [DI]



EA = (DS)₀ + DI = XXXXX →

BX	60	50
DS	10	70

← يتم تحديد بدايه العنوان من EA

← نأخذ من memory بيانات تحجها بحجم المكان التى سوف توضع به

Hinit

byte = 2^bit / memory size

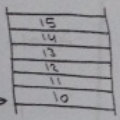
في المثال BX حجمه 16 bit اى 2 byte : نأخذ مكانيم من الذاكره ونضعهم في BX بحيث يوضع العنوان الأول من الذاكره في low byte والمكان الثاني في high byte

← ثم نأخذ العنوانيم التاليم ونضعهم في DS حيث آن الأمر LDS

→ LSS

ex: lss EAX, [BX+DI]

EA = (DS)₀ + BX + DI = XXXX →



EAX	13	12	11	10

where EAX → 32 bit

SS	15	14

→ LES

ex: lES BX, [DI]

EA = (DS)₀ + DI = XXXXX →

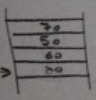
BX	60	50

, ES | 80 | 70 |

→ LFS

ex: LFS BX, [DI]

EA = (DS)₀ + DI = XXXXX →



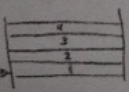
BX	60	80

, FS | 70 | 50 |

→ LGS

ex: LGS BX, [DI]

EA = (DS)₀ + DI = XXXXX →



BX | 2 | 1 | , GS | 4 | 3 |

LCS EAX, [DI] (X)

because CS mustn't be Changed.

String Instructions

LODS, STOS, MOVS, INS, OUTS

This Instructions depend on direction

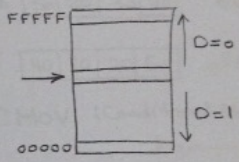
	D	FLAG

D=0 ⇝ autoincrement mode

← يسير في اتجاه زياده العنوان

D=1 ⇝ autodecrement mode

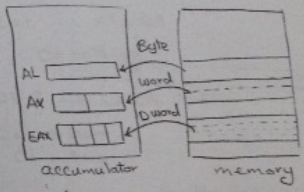
← يسير في اتجاه نقص العنوان



[1] LODS

accumulator ← memory نأخذ من

ثم أزود أو أقلل حسب D offset



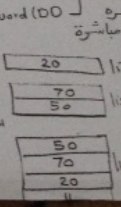
LODSB ⟶ byte
LODSW ⟶ word
LODSD ⟶ double word

← المكان غير معروف مباشره بوصله EA عن طريق

LODS list ⟶ byte (DB)
LODS Outer ⟶ word (DW)
LODS FROG ⟶ double word (DD)

← المكان الذي يتم وضع فيه في الذاكره معروف مباشرة

list: DB 20H
list2: DW 5070H
list3: DD 50702011H



and can be Signed integer (IMUL) or Unsigned

From Source → to Destination

Data Segment — offset SI

Extra Segment — DI

ex: DS = 1000H , SI = 1000H , D = 0
execute LODSW then LODSD

Solution

EA = (DS)0 + SI = 10000 + 1000 = 11000

$SI_{new} = SI_{old} + 2\,bit = 1002$

AX | 90 | 50 |

	5
	6
	5
11002 →	10
	90
11000 →	50

D=0

To execute LODSD

SI = 1002 , DS = 1000 , D = 0

EA = 10000 + 1002 = 11002

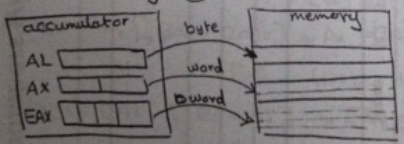
EAX | 7 | 6 | 5 | 10 |

$SI_{new} = SI_{old} + 4\,bit = 1002 + 4 = 1006$

If D=1 $SI_{new} = SI_{old} -$ (1 → byte, 2 → word, 4 → Dword)

2 STOS

يأخذ من accumulator ويضع في memory



D ثم يزود أو يقلل offset بـ

STOSB , STOSW , STOSD
STOS list (DB) , STOS list2 (DW)
STOS list2 (DD)

3 MOVZX

Register زود باقي خانات بـ Zeros اصفار

MOVSX (Memory) زود باقي المساحة بنفس القيمة في الإشارة

4 XCHG

بدل المكان

→ Both places must have the same SIZ

→ CS mustn't be Change

Examples:

XCHG AL, CL (✓)
XCHG CX, BP (✓)
XCHG Data, AX (✓)
XCHG DS, ES (X)
XCHG CS, AX (X)

5 BSwap

→ work only with 32 bit registers (80486 → P3, P4)

EAX | 50 | 20 | 30 | 40 | before

EAX | 40 | 30 | 20 | 50 | after

6 CMOV (Conditional Mov)

نقل مشروط

CMOVZ → هذا الأمر معناه
1- يرى خانة Z في Flag
Z = 0 لا يفعل شيء
Z = 1 ينقل

ex: CMOVZ AX, BX
If Z=1 ∴ AX = BX
else Do nothing

→ CMOVC → Chick Carry
If C=1 → Move
else Do nothing

→ CMOVNC → Chick No Carry
If C=0 → Move
else Do nothing

Ch 5 Arithematic & Logic Instruction

[1] Arithematic

binary operator:
- ADD
- ADC
- SUB
- SBB

unary operator:
- INC
- DEC

- MUL
- IMUL
- DIV
- IDIV
- CMP
- JA, JB, JAE
- JBE, JE, JNE

[2] Logic

- AND
- OR
- NOT
- NEGATE
- TEST
- XOR
- BT
- BTC
- BTS
- BTR
- Shift → (logical) unsigned numbers → SHL (*2), SHR (/2)
- Shift → (Arithematic) signed numbers → SAL, SAR
- Rotate → without Carry → ROR, ROL; with Carry → RCR, RCL

→ Arithematic

① ADD

→ ADD AL, 20H → $AL_{new} = AL_{old} + 20$ [Immediate Addressing mode] EA = No E.A

→ ADD AL, BH → $AL_{new} = AL_{old} + BH$ [Register Addressing mode] E.A = BH

→ ADD AL, NUM → $AL_{new} = AL_{old} + NUM$ [Direct Addressing mode] EA = NUM

→ ADD AL, [BX] → $AL_{new} = AL_{old}$ + place in memory (1 byte) [Register Indirect Addressing Mode]

EA of [BX] = $(DS)_0 + BX$ ⟶

→ ADD AX, [BX + 2000] → $AX_{new} = AX_{old}$ + place in memory (2 byte) [Register - Relative Addressing Mode]

$EA = (DS)_0 + BX + 2000$

→ ADD EAX, [BP + SI] → $EAX_{new} = EAX_{old}$ + place in memory (4 byte) [Bass plus Index Addressing Mode]

$EA = (SS)_0 + BP + SI$

→ ADD AX, 2000 [BX + SI] → $AX_{new} = AX_{old}$ + place in memory (2 byte) [Base relative plus Index Addressing Mode]

$EA = (DS)_0 + BX + SI + 2000$

→ ADD 2000 [BX + SI], EAX ⟶ place in memory (4 byte) = place in memory (4 byte) + EAX

EA = $(DS)_0$ + BX + SI + 2000

V.I

① ADD ES, DS (X)

② ADD [BX], [BP] (X)

③ ADD CS, AX (X)

④ ADD AX, CS (✓)

⑤ ADD CX, BL (X)

(Compare) → We Compare between things to make another ...

Carry → C

Bx Bx +

Bx Cx

Dx Ax

② ADC (add with Carry)

→ ADC AL, 20H → $AL_{new} = AL_{old} + 20_H + C$

وبأخذ باقي أشكال Addressing Modes مثل الأمثلة السابقة لـ ADD

③ SUB

→ SUB AL, 20H → $AL_{new} = AL_{old} - 20_H$

وأيضا هو وباقي Instructions تأخذ نفس أشكال Addressing Modes مثل الأمثلة السابقة لـ ADD

④ SBB (Sub with Borrow)

Borrow : Flag في Carry خانة نفس هي

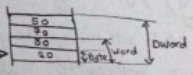
→ SBB AL, 20H → $AL_{new} = AL_{old} - 20_H - C$

⑤ INC (Add 1)

→ INC AX → $AX_{new} = AX_{old} + 1$

V.I → INC Byte ptr / Word ptr / Dword ptr [Bx]

EA = (DS)0 + Bx



→ Byte 20 + 1 = 21

→ Word 80 20 + 1 = 80 21

→ Dword 50 70 80 20 + 1 = 50 70 80 21

→ INC ES (X)

→ INC CS (X)

Operators

Unary: INC, DEC — X++

Binary: ADD, SUB — X+=Y

Ternaly: X > 3 ? Yes: Y=10 / NO: Y=18

⑥ DEC (Sub 1)

→ DEC AX → $AX_{new} = AX_{old} - 1$

→ DEC ES (X)

→ DEC CS (X)

Multiplication is performed on bytes, words or doublewords, and can be Signed integer (IMUL) or Unsigned integer (MUL). The product after a multiplication is always a double-width product.

	Object / type	Examples:
(7) MUL The Multiplicand is always in Accumulator So the multiplication instruction contains one operand.	(a) 8 bit Multiplication – The multiplicand is always in the AL register – The multiplier can be 8-bit register or any memory location. – After the multiplication, the unsigned product is placed in AX a double-width product	MUL BL MUL CL MUL Temp
	(b) 16 bit Multiplication – The multiplicand is always in the AX register – The multiplier can be 16 bit register or any memory location – After the multiplication the unsigned product is placed in EAX a double-width product.	MUL CX MUL word ptr [SI]
	(c) 32 bit Multiplication – Only the 80386 through P4 processors multiply 32 bit doubleword. – The multiplicand is always in the EAX register – The multiplier can be 16 bit register or any memory location – The product (64 bit wide) is found in EDX-EAX where EAX contains the least-significant 32 bit of the product.	MUL ECX MUL Dword ptr [ECX]
(8) IMUL The Multiplicand is always in Accumulator – The product is in true binary form. If positive, and in two's comp if negative. – These are the same forms used to store all +ve and –ve Signed numbers used in micropro-	(a) 8 bit Multiplication – AL is multiplied by 8-bit register (ex: DH) or any memory location (ex: [BX]), the signed product is in AX	IMUL DH IMUL byte ptr [BX]
	(b) 16 bit Multiplication – AX is multiplied by 16 bit register (ex: DI), – The signed product is in DX - AX	IMUL DI
	(c) 32 bit Multiplication – EAX is multiplied by 32 bit register (ex: EDI) – The signed product is in EDX-EAX → Only the 80386 P4 processors multiply 32 bit double word.	IMUL EDI

(11) CMP

Division occurs on 8- or 16 bit numbers in the 8086 - 80286 proc
and on 32 bit numbers in the 80386 P4. These numbers
signed (IDIV) or unsigned (DIV) integers. The dividend is alway
a double-width dividend that is divided by the operand.
There is no immediate division instruction any micro

	Object / type	Example
(9) DIV The Accumulator is used to store the dividend.	ⓐ 8-bit Division – AX register to store the dividend – The dividend is divided by the contents of any 8 bit register or memory location. – The quotient moves into AL after the division with AH containing a whole number remainder.	DIV CL DIV Byte ptr [BE
	ⓑ 16-bit Division – is similar to 8 bit division except that instead of dividing into AX, the 16-bit number is divided into DX - AX, a 32 bit dividend	DIV CX DIV NUMB
	ⓒ 32 bit Division – The 64-bit contents of EDX - EAX are divided by the operand specified by the instruction, leaving a 32-bit quotient in EAX and a 32 bit remainder in EDX	DIV ECX DIV DATA2
(10) IDIV	ⓐ 8-bit Division – AX is divided by 8 bit register (ex BL), the signed quotient is in AL and the remainder is in AH.	IDIV BL
	ⓑ 16-bit Division – DX-AX is divided by 16-bit register (ex SI), the signed quotient is in AX and the remainder is in DX	IDIV SI
	ⓒ 32 bit Division – EDX-EAX is divided by the doubleword contents of the data segment memory location addressed by EDI; the signed quotient is in EAX and the remainder is in EAX	IDIV DWORD PTR [E

IMP

⑪ CMP (Compare)

→ We Compare between things to make another operations depend on the relation (<, =, >) between this things.

→ CMP AL, BH

JA → A>B
JB → B>A
JAE → A≥B
JBE → B≥A
JE → A=B
JNE → A≠B

① AL − BH
- AL > BH → Result +ve Then S=0, Z=0
- AL = BH → Result 0 Then Z=1
- AL < BH → Result −ve Then S=1, Z=0

Note: This operation changes S and Z in Flag

→ CMP AL, BH
JA Lable1
JB Lable2
other instructions
Lable1: ...
Lable2: ...

① AL − BH
a) Result +ve ⇝ AL > BH
- Jumb to Lable 1 to do its operations, neglecting JB lable2 and the other instructions.

b) Result −ve ⇝ BH > AL
- Jumb to Lable 2 to do its operations, neglecting the other instructions.

c) if AL isnot > BH or BH > AL
- go to do other instructions.

→ CMP [AL], [BH] (X)
→ CMP CS, SS (X)
→ CMP CS, Ax (✓)
→ CMP Ax, AL (X)

→ Logic

① AND

→ AND AL, 20H

assume AL = 10H

AL = 00010000
20_H = 00100000 and
AL_{new} = 00000000 = 0H

② OR

→ OR AL, 20H

AL = 10H

AL = 00010000
20_H = 00100000 OR
AL_{new} = 00110000 = 30_H

③ NOT (1's complement)

→ NOT Byte ptr / word ptr / Dword ptr [Bx]

1's Complement → 0 ⇝ 1, 1 ⇝ 0

④ NEGATE (2's complement)

Note: 2's Complement

نمشي من اليمين ننزل 0 وأول واحد 1 يقابلنا ننزله وبعده نقلب 0 اى 1 والعكس

ex: 111010 , 001
2's Complement: 000110 , 111

④ TEST

يختبر الوزن الموضعي (١٠٥) لخانه يحدد رقمها

وبالتالي يليه امر آخر JZ , JNZ

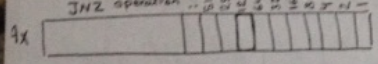
JZ → Jump if 0

JNZ → Jump if no zero

→ TEST Ax , 128

JZ operation

JNZ operation



← result ?

---

⑤ BT

يختبر bit واحده (موجود فقط في 80386-P4)

→ BT AX , 4

Test bit position 4 in Ax.

---

⑥ BTC

→ test and complement

→ BTC Ax , 128

JZ op1

JNZ op2

← يقوم بعمل Test لـ bit واحده

ثم [1] اذا كانت بـ 0 يحولها الى 1

[2] اذا كانت بـ 1 يحولها الى 0

---

⑦ BTS

→ Test and Set

→ BTS Ax, 4

Test and set bit position 4 in Ax

← يقوم بعمل Test لـ bit واحده

ثم يضع بها 1

---

⑧ BTR

→ Test and Reset

→ BTR Ax, 4

Test and Reset bit position 4 in Ax

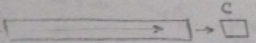
← يقوم بعمل Test لـ bit واحده

ثم يضع بها 0

(11) Shift : Shift instructions position or move numbers to the left or right within a register or memory location.

[1] Unsigned (logical) numbers

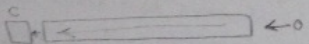
→ Shift Right SHR



Ex: SHR, BX, 12 (BX is logically shifted right 12 places)

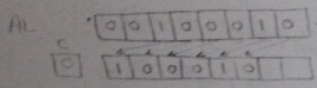
SHR ECX, 10 (ECX is logically shifted right 10 places)

→ Shift Left SHL



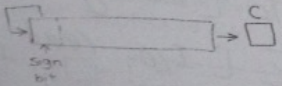
Ex: SHL AX, 2 (AX is logically shifted left 2 places)

SHL AL, 2 assume AL = 22H



[2] Signed (Arithematic) numbers

→ Shift Right SAR



← في هذا النوع لابد من الحفاظ على الاشاره ثابته

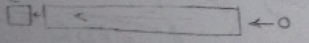
Ex: SAR SI, 2

(SI is arithmetically shifted right 2 places)

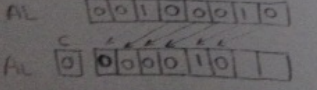
SAR EDX, 14

(EDX is arithmetically shifted right 14 places)

→ Shift Left SAL



SAL AL, 2 assume AL = 22H

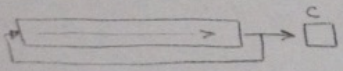


(17) Rotate

the information in a register or memory location, either from one end to another or through the Carry Flag.

[1] Rotate without Carry

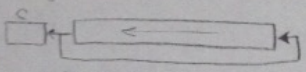
→ Rotate Right ROR



Ex: ROR WORD PTR [BP], 2

(The word contents of the stack segment memory location addressed by BP rotate right 2 places)

→ Rotate Left ROL

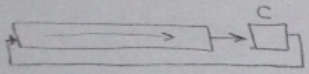


Ex: ROL SI, 14 (SI rotates left 14 places)

ROL ECX, 18 (ECX rotates left 18 places)

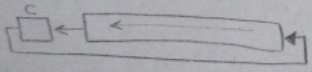
[2] Rotate With Carry

→ Rotate Right RCR



Ex: RCR AH, CL (AH rotates right through Carry the number of places specified by CL)

→ Rotate Left RCL



Ex: RCL BL, 6 (BL rotates left Through Carry 6 places)